*Compiling Input** FO(·) *Inductive Definitions into Tabled Prolog Rules for* IDP3

JOACHIM JANSEN, ALBERT JORISSEN, GERDA JANSSENS

Department of Computer Science, KU Leuven

(*e-mail:* firstname.secondname@cs.kuleuven.be, albert.jorissen@ulyssis.org)

submitted 10 April 2013; revised 23 May 2013; accepted 23 June 2013

Abstract

FO(·)$^{\text{IDP3}}$ extends first-order logic with inductive definitions, partial functions, types and aggregates. Its model generator IDP3 first grounds the theory and then uses search to find the models. The grounder uses Lifted Unit Propagation (LUP) to reduce the size of the groundings of problem specifications in IDP3. LUP is in general very effective, but performs poorly on definitions of predicates whose two-valued interpretation can be computed from data in the input structure. To solve this problem, a preprocessing step is introduced that converts such definitions to Prolog code and uses `XSB` Prolog to compute their interpretation. The interpretation of these predicates is then added to the input structure, their definitions are removed from the theory and further processing is done by the standard IDP3 system. Experimental results show the effectiveness of our method.

KEYWORDS: program transformation, FO(·), logic programming, tabling, knowledge base systems, IDP system, declarative modeling

1 Introduction

Recent proposals for declarative modeling use first-order logic as their starting point. Examples are Enfragmo (Aavani et al. 2012) and FO(·)$^{\text{IDP3}}$, the instance of the FO(·) family that is supported by IDP3, the current version of the IDP Knowledge Base System (De Pooter et al. 2011). FO(·)$^{\text{IDP3}}$ extends First-Order (FO) logic with inductive definitions, partial functions, types and aggregates. IDP3 supports model generation and model expansion (Mitchell and Ternovska 2005; Wittocx et al. 2013) as inference methods.

IDP3 supports these inference methods using the ground-and-solve approach. First the problem is grounded into an Extended CNF (ECNF) theory. Next a SAT-solver is used to calculate a model of the propositional theory. One of the problems with this approach is the possible combinatorial blowup of the grounding. A predicate $p(x_1, x_2 \dots x_n)$ with s as the size of the domain of its arguments has s^n possible instances. A grounding that has to represent all these possible instances is therefore possibly very large.

Most Answer Set Programming (ASP) systems solve this problem by using semi-naive bottom-up evaluation (Faber et al. 2012; Gebser et al. 2011) with optimizations, which in particular is very effective in dealing with large data sets and intentional predicates defined in terms of these. However, the goal of FO(·) is to offer a very high level language for

complex (NP) search problems - hence FO(·) bodies in definitions and FO(·) constraints - and to let users concentrate on the logic specifications of problems rather than having to fine-tune the specification in order to obtain a compact grounding. The grounder of IDP3 uses a Lifted Unit Propagation (LUP) method to derive additional information for subformulas: bounds for certainly true, certainly false and unknown (Wittocx et al. 2010). This grounding approach is very good for NP search problems, but does not deal adequately with large data sets and intentional predicates defined in terms of these.

In this paper we propose an independent (pre-processing) step that reduces the IDP3 grounding by calculating in advance the two-valued interpretations of such predicates. We transform their first-order definitions into Prolog rules and use a Prolog system with tabling to compute the interpretations. We decided to use `XSB` because we also plan to explore its tabling support for further improvements of the grounder. Experimental results show the effectiveness of our method.

In Section 2 we introduce IDP3 and FO(·). In Section 3 we describe the transformation of FO(·) definitions to Prolog rules. In Section 4 we discuss the current integration of IDP3 with `XSB` and further extensions of using `XSB` together with IDP3. In Section 5 we present experimental results. Section 6 contains related work and concludes.

2 Terminology and Motivation

2.1 The IDP3 *System*

We focus on the aspects of $\text{FO}(\cdot)^{\text{IDP3}}$ that are relevant for this paper. More details can be found in (De Pooter et al. 2011) and (Blockeel et al. 2012) where one can find several examples. An $\text{FO}(\cdot)^{\text{IDP3}}$ model consists of a number of logical components, namely vocabularies, structures, terms, and theories. A *vocabulary* declares the symbols to be used. A *structure* is used to specify the domain and data; it provides a partial (possibly three-valued) interpretation of the symbols in the vocabulary. A *theory* consists of $\text{FO}(\cdot)^{\text{IDP3}}$ formulas and definitions.

A *definition* is a set of *rules* of the form $\forall \bar{x} : p(\bar{x}) \leftarrow \phi[\bar{x}]$. where $\phi[\bar{x}]$ is an $\text{FO}(\cdot)^{\text{IDP3}}$ formula. An $\text{FO}(\cdot)^{\text{IDP3}}$ *formula* differs from FO formulas in two ways. Firstly, $\text{FO}(\cdot)^{\text{IDP3}}$ is a many-sorted logic: every variable has an associated *type* and every type an associated domain. Moreover, it is order-sorted: types can be subtypes of others. Secondly, besides the standard terms in FO, $\text{FO}(\cdot)^{\text{IDP3}}$ formulas can also have aggregate terms: functions over a set of domain elements and associated numeric values which map to the sum, product, cardinality, maximum or minimum value of the set.

The model expansion of IDP3 extends a partial structure (an interpretation) into a two-valued structure that satisfies all constraints specified by the $\text{FO}(\cdot)^{\text{IDP3}}$ model. Formally, the task of model expansion is, given a vocabulary V, a theory T over V and a partial structure S over V (at least interpreting all types), to find a two-valued structure $\mathcal{M}$ that satisfies T and extends S, i.e., $\mathcal{M}$ is a model of the theory and the input structure S is a subset of $\mathcal{M}$. As an illustration, we give the $\text{FO}(\cdot)^{\text{IDP3}}$ model of the NQueens problem in Figure 1. The NQueens problem consists of placing n queens on a n `x` n chess board in such a way that the queens cannot attack each other.

The *vocabulary* V consists of a two types (*diag* and *index*), one constant (n), one predicate (*queen*), and two functions (*diag*1 and *diag*2). The *structure* S provides a two-valued

vocabulary V {
  type *index* isa int
  $queen(index, index)$
  $n : index$
  type *diag* isa int
  $diag1(index, index) : diag$
  $diag2(index, index) : diag$
}
structure S : V {
  $index = \{1..4\}$
  $diag = \{1..7\}$
  $n = 4$
}
theory T : V {
  $\{diag1(x, y) = d \leftarrow d = x - y + n.\}$
  $\{diag2(x, y) = d \leftarrow d = x + y - 1.\}$

  $\forall x[index] : \exists_{=1} y[index] : queen(x, y).$ // (1)
  $\forall y : \exists_{=1} x : queen(x, y).$ // (2)

  $\forall d : \#\{xy : queen(x, y) \wedge diag1(x, y) = d\} < 2.$ // (3)
  $\forall d : \#\{xy : queen(x, y) \wedge diag2(x, y) = d\} < 2.$ // (4)
}

Fig. 1. An FO$(\cdot)^{\text{IDP3}}$ model for NQueens with n = 4

interpretation for the constant $n/0$ (the size of the n x n board) and the *diag* and *index* types that follow from this. The predicate $queen(index, index)$ represents where queens are placed on the board. The function $diag1(index, index) : diag$ maps each square on the board to its upper-left-to-lower-right diagonal. The function $diag2(index, index) : diag$ is defined similarly for the lower-left-to-upper-right diagonals. The definitions for $diag1$ and $diag2$ are placed in *theory* T and are given between "{" and "}".

The theory also specifies the constraints expressing that the placed queens cannot attack each other. The first constraint expresses that there is exactly one queen on each column of the board: for every x of type *index* ($\forall x[index]$), there is exactly one y of type *index* ($\exists_{=1} y[index]$) for which $queen(x, y)$ holds. The quantified variables are typed in this constraint. This typing is optional and has been omitted in the other constraints as IDP3 can derive types from the information in the vocabulary. The second constraint expresses that there is exactly one queen on each row of the board. The third constraint expresses that for each diagonal ($\forall d$) there are less than two elements in the set $\{x\ y : queen(x, y) \wedge diag1(x, y) = d\}$; the set of elements (x, y) such that they satisfy $queen(x, y) \wedge diag1(x, y) = d$. This ensures that there is no more than one queen on each upper-left-to-lower-right diagonal. The fourth constraint expresses the same for the lower-left-to-upper-right diagonals.

The IDP3 system performs model expansion by first reducing the problem to Extended CNF, using the grounder GIDL (Wittocx et al. 2010), and subsequently calling the solver MINISAT(ID) (Mariën et al. 2008; Eén and Sörensson 2003). In this case, model expansion results in two-valued interpretations for *queen*, $diag1$ and $diag2$.

2.2 Input, Open, and Input* Predicates

Consider the set of rules in the definitions of an FO($\cdot$)$^{\texttt{IDP3}}$ model. The *defined* predicates[1] are the predicates that appear in a head of a rule. The other predicates, which appear only in the bodies, are the *open* predicates. An *input* predicate is a predicate for which the structure S specifies the two-valued interpretation. It can be an open or a defined predicate.

Given a rule $\forall \bar{x} : p(\bar{x}) \leftarrow \phi[\bar{x}]$., we say that $p(\bar{x})$ *depends on* $q(\overline{x})$ if the predicate $q(\overline{x})$ appears in $\phi[\bar{x}]$. We compute the *depends on* relation for all the defined predicates and construct the dependency graph. The leaves in this graph are open symbols (predicates or functions).

Some defined predicates can be calculated in advance. We call these predicates the *input*$*$ predicates and define them to be the *defined* predicates that are not *search* predicates. A predicate p is a *search* predicate if

- predicate p is an open predicate but not an input predicate
- predicate p depends on a predicate q and q is a *search* predicate.

We propose to evaluate these input$*$ predicates before grounding and solving. Because FO($\cdot$)$^{\texttt{IDP3}}$ definitions have FO($\cdot$)$^{\texttt{IDP3}}$ formulas as bodies, which are obviously not valid Prolog bodies, transformations of these FO($\cdot$)$^{\texttt{IDP3}}$ formulas into Prolog bodies are needed. Our proposed method applies three standard steps on the FO($\cdot$)$^{\texttt{IDP3}}$ model:

1. Detection of the input$*$ predicates in the theory T and the structure[2] S
2. Transformation of the definitions of the input$*$ predicates into Prolog rules
3. Computation of the two-valued interpretation of the input$*$ predicates with `XSB` using these Prolog rules

The result of our method is a structure S' that extends S with two-valued interpretations for the input$*$ predicates and a theory T' with all constraints from T and all but the input$*$ definitions from T. Grounding now starts from a more precise structure and a smaller theory.

2.3 An Example for NQueens with $n = 4$

In Figure 1, *diag*1 and *diag*2 are input$*$ predicates[2] because they do not depend on any *search* predicate: n is an open input predicate given in the structure and built-ins ($+$ and $-$) are always input predicates. The definitions of *diag*1 and *diag*2 both have three variables. Two variables range over *index* and one ranges over *diag*. This means that the number of instances created by naive grounding is of the order n^3. For the structure in Figure 1 where $n = 4$, this results in $4^2 * 7 = 112$ instances per definition, resulting in 224 instances in total.

Our method calculates the interpretation of the input$*$ predicates *diag*1 and *diag*2 by executing the following `XSB` program:

[1] We consider constants and functions as a special case of predicates, unless we explicitly make a distinction between them.

[2] In this paper we assume that input$*$ predicates have no given interpretation in the input structure and are only defined in one definition.

```
:- table diag1/3, diag2/3.
diag1(X,Y,D) :- n(N), type_index(X), type_index(Y), D is X - Y + N, type_diag(D).
diag2(X,Y,D) :- type_index(X), type_index(Y), D is X + Y - 1, type_diag(D).

type_diag(X) :- between(1,7,X).
type_index(X) :- between(1,4,X).
n(4).

?- diag1(X,Y,D).
?- diag2(X,Y,D).
```

The two queries have variables as arguments. The answer `diag1(`i_x`,`i_y`,`d`)` means that $diag1(i_x, i_y) = d$. The `XSB` program calculates two diagonals for each combination of two indices. The number of resulting answer tuples is n^2 for each diagonal (32 tuples in total for $N = 4$). These tuples are added as interpretation for the input$*$ predicates. In this paper we generate this method for FO$(\cdot)^{\texttt{IDP3}}$ definitions of the form $\forall \bar{x} : p(\bar{x}) \leftarrow \phi[\bar{x}]$.

3 Transformation

In order to use Prolog for the calculation, we need to transform the relevant part of the FO$(\cdot)^{\texttt{IDP3}}$ model into Prolog, namely the definitions of the input$*$ predicates and the interpretation of the open predicates (including types). The resulting Prolog program is correct if the interpretation that it calculates for the input$*$ predicates is exactly the same as the interpretation that IDP3 would find without `XSB`.

3.1 Transforming the Structure

First we transform the relevant information in the structure part of the FO$(\cdot)^{\texttt{IDP3}}$ model. For every true tuple $(t_1, t_2 ... t_n)$ in the interpretation of an open input predicate o, we add a fact `o(`t_1`,`$t_2 ... t_n$`).` to the Prolog program. If o is a function symbol, we use its predicate representation (see Section 3.2.1).

We also transform all types used in the definitions of the input$*$ predicates. Each type is given by a domain. We generate the corresponding *type predicates*. For each type t, a new Prolog predicate `type_t` is added to the Prolog program.
If type t has a numerical domain ranging from *lower_bound* to *upper_bound* (e.g. index = {1..4}), we add `type_t(X) :- between(`*lower_bound*`,`*upper_bound*`,X).`

If type t has an enumerated domain (e.g. node = {a,b,c,d}), we add `type_t(`*dom*`).` for every domain element *dom*. Note that an enumerated domain of size n results in n facts, whereas a numerical ranged domain results in one rule.

We need to transform these types and add the type predicates to `XSB` for of two reasons:

- An interpretation needs to be well-typed. The type predicates are used for type *checking* before answers are returned.
- Sometimes the values of the arguments of a predicate need to be known (for example, negation as failure is only safe if its goal is ground). We use the type predicates of the type of the variables as *generators* of their values.

3.2 Transforming the Theory

3.2.1 Simplifying the FO(·) definitions

Eliminating functions We transform all functions into predicate form: $f(\bar{x}) = t$ is transformed into $p_f(\bar{x}, y) \ \wedge \ y = t$. We also take notice of the constraint that there exists exactly one y for every tuple $\bar{x}$ such that $p_f(\bar{x}, y)$ holds. When IDP3 converts the answer tuples to its internal representation, this constraint is checked. Any atom $p(f(\bar{x}))$ that has a function result as argument, is replaced with $p(u)$ and we add $p_f(\bar{x}, u)$ to the body of the definition. Doing this we assure that the arguments of the atoms are variables.

Eliminating equivalences Definitions with equivalences in their body are rewritten to their implicational form.

Eliminating implications Definitions with implications in their body are rewritten to their disjunctive form.

Pushing down negations Negations are pushed into subformulas until they are applied directly to atomic formulas.

Flattening out nested conjunctions and disjunctions Redundant parenthesis of nested conjunctions and disjunctions are removed.

3.2.2 Predicate introduction

Next we use predicate introduction (Wittocx 2010) to simplify the bodies of the definitions into elementary formulas. An elementary formula is a formula where all subformulas are atomic formulas. An atomic formula is either an atom, a negated atom or a numerical expression (i.e., an IDP3 built-in). We use the following rewrite rule:

1. For some non-atomic subformula $\psi[\bar{x}]$ that occurs inside a conjunction, disjunction or quantified formula of $\phi[\bar{x}]$, introduce a new predicate p_ψ of the same arity as $\psi[\bar{x}]$
2. Substitute $p_\psi(\bar{x})$ for $\psi[\bar{x}]$ in $\phi[\bar{x}]$
3. Add the rule $\forall \bar{x} : p_\psi(\bar{x}) \leftarrow \psi[\bar{x}]$.

This rewrite rule is applied until all definitions have only elementary formulas as body. These transformations are proven to result in logically equivalent definitions. After predicate introduction, the rules in the definitions have one of the following four forms, where $\bar{x}$ is the union of $\bar{x}_1 \ldots \bar{x}_n$:

$$\forall \bar{x} : p(\bar{x}) \leftarrow p_{\phi_1}(\bar{x}_1) \wedge p_{\phi_2}(\bar{x}_2) \wedge \ldots \wedge p_{\phi_n}(\bar{x}_n).$$

$$\forall \bar{x} : p(\bar{x}) \leftarrow p_{\phi_1}(\bar{x}_1) \vee p_{\phi_2}(\bar{x}_2) \vee \ldots \vee p_{\phi_n}(\bar{x}_n).$$

$$\forall \bar{x}_1 : p(\bar{x}_1) \leftarrow \exists \bar{x}_2 : p_\phi(\bar{x})$$

$$\forall \bar{x}_1 : p(\bar{x}_1) \leftarrow \forall \bar{x}_2 : p_\phi(\bar{x})$$

3.2.3 To-Prolog

We start with the transformation of the four kinds of rules and continue with the atomic formulas. We table the input∗ predicates and use `XSB` to compute the answers according to the well-founded semantics using the transformed code.

We use the following invariant to prove that the to-Prolog transformations are correct: after executing a Prolog (sub)goal `p_phi`, all its arguments are grounded and equal to the interpretation of the corresponding logical predicate p_ϕ. This invariant will be proven for each transformation in this subsection.

Unsafe variables The variables of a Prolog clause that do not occur in a positive goal in the body of the clause or that only occur as input arguments of Prolog built-ins, are so-called *unsafe* variables. The call `typesunsafe(U)` is a shorthand for `type_t1(U1), type_t2(U2) ... type_tn(Un)`, with `U1 ... Un` the unsafe variables of the clause. We add `typesunsafe(U)` in the body of the clause to use their type predicates as generators.

Conjunction

$$\forall \bar{x} : p(\bar{x}) \leftarrow p_{\phi_1}(\bar{x}_1) \wedge p_{\phi_2}(\bar{x}_2) \wedge ... \wedge p_{\phi_n}(\bar{x}_n).$$

results in

```
p(X) :- typesunsafe(U),p_phi_1(X_1), p_phi_2(X_2) ... p_phi_n(X_n).
```

The set `U` denotes the unsafe variables of the clause. To improve performance and to avoid errors due to non-ground negative calls or built-ins for which an input argument is non-ground, the bodies are reordered according to a simple form of query optimization based on the following rules:

1. move negative literals as much as possible to the left (but each variable of the negative literal must have an occurrence in a positive call to its left)
2. move built-ins as much as possible to the left (but each input variable of the built-in must have an occurrence in a positive call to its left)
3. move positive calls to tabled predicates (input* predicates) to the left of introduced predicates this results in less different call patterns and better performance (Swift and Warren 2012).
4. move the `type_t` calls of the unsafe variables that occur only in the head to the end of the body.
5. put open input predicates before input$*$ predicates to bind variables as soon as possible.

This reordering is also done for the other three kinds of rules.

Our invariant is satisfied because `p(X)` succeeds only if all the subgoals succeed, all the variables get ground values and the `p_phi_j(X_j)` goals satisfy the invariant. Thus, all variables of `p(X)` are instantiated with the correct values.

Disjunction

$$\forall \bar{x} : p(\bar{x}) \leftarrow p_{\phi_1}(\bar{x}_1) \vee p_{\phi_2}(\bar{x}_2) \vee ... \vee p_{\phi_n}(\bar{x}_n).$$

results in

```
p(X) :- typesunsafe(U_1), p_phi_1(X_1).
p(X) :- typesunsafe(U_2), p_phi_2(X_2).
                    ...
p(X) :- typesunsafe(U_n), p_phi_n(X_n).
```

Our invariant is satisfied because if `p(X)` succeeds, one of its rules succeeds. The

`p_phi_j(X_j)` goals satisfy the invariant and with the addition of the type predicates, all returned variables are bound correctly.

Existential

$$\forall \bar{x}_1 : p(\bar{x}_1) \leftarrow \exists \bar{x}_2 : p_\phi(\bar{x})$$

results in

```
p(X_1) :- typesunsafe(U), p_phi(X).
```

Our invariant is satisfied because `p(X_1)` succeeds only if `p_phi(X)` succeeds, `p_phi(X)` returns the correct values, and with the addition of the type predicates, all returned variables are bound correctly.

Universal

$$\forall \bar{x}_1 : p(\bar{x}_1) \leftarrow \forall \bar{x}_2 : p_\phi(\bar{x})$$

results in

```
p(X_1) :- type_t(X_1), idp_forall(type_t(X_2), p_phi(X)).
```

The `X_1` variables that appear in `p_phi(X)` will be grounded by their `type_t` generators before the call to `idp_forall`. The `X_2` are grounded in the first argument of `idp_forall`. The Prolog predicate `idp_forall(C1,C2)` is defined such that it only succeeds if for every succeeding call `C1`, call `C2` also succeeds:

```
idp_forall(C1,C2) :- call_tv(tables:not_exists((call(C1),
                     tables:not_exists(C2))),true).
```

Two built-in predicates are used: `call_tv/2` is used to filter out undefined answers and `not_exists/1` is used because it does negation for a call of which it is not known whether it is a tabled or not (Swift et al. 2013). The invariant is satisfied due to the `type_t` generators and the semantics of `idp_forall/2`, `not_exists/1` and `call_tv/2`.

Atomic formulas The atomic formulas in the elementary formulas consist of two kinds:

1. Atomic formulas $p(t_1, \ldots t_n)$ with p an open input, input$*$, or introduced predicate. Results in `p(T1,...Tn)`, with `Ti` the variable representing t_i. If this atomic formula is negated, `tnot/1` (for tabled predicates) or `"\+/1"` (for other predicates) is used.
2. Atomic formulas consisting of numerical operations ($+, *$...) or comparisons ($<, >$ ...). Prolog has built-ins for each operation and comparison. We add type checks for the output arguments of these built-ins.

In case of an input predicate, the call `p(T1,T2...Tn)` returns as answers all the tuples for which p is true. This is due to our transformation of the input predicates in the structure. In case of a negated atom, it is assured to be ground by our `typesunsafe` calls and our body reordering. For a ground body, negation as failure executes as logical negation. For the `tnot` goals we rely on the dynamic stratification of `XSB`. Finally, the arithmetic expressions are correctly computed because of our `typesunsafe` calls, our body reordering and because our type checking ensures that the ouput arguments of Prolog built-ins have correct types.

Aggregate terms An aggregate term is a term that is the result of an aggregate function. An aggregate function maps a set to a single element. Examples of aggregate functions are: cardinality (how many elements are there in the set), minimum, maximum...

Aggregate terms can still be present as arguments in the atomic numerical formulas of above. Due to space limitations and the technical nature of this subject, we present the the transformation of aggregate terms in the appendix.

4 Computing Interpretations with XSB

4.1 About Using XSB *for input∗ predicates*

Like other Prolog implementations, XSB provides support for both statically compiled code and dynamically asserted code. The transformed rules are compiled and loaded using the `consult/1` command. Enumerated facts are loaded dynamically using the `load_dync/1` command. We do this because for large files containing $10^4 - 10^7$ facts, dynamic loading is much faster than XSB's compiler (Swift et al. 2013).

For every input∗ predicate $p(x_1, x_2..x_n)$ we table the answers for the query `p(X1,X2..Xn)` using local scheduling (and the default variant tabling) of XSB. For recursive predicates we rely on the tabling to detect loops.

Once the answers are computed, they are used in IDP3 to construct the interpretation of the corresponding predicate (or function). The definitions of the input∗ predicates are removed from the theory. In between separate executions of this process, XSB has to be 'reset': we need to empty the tables and remove the loaded Prolog code.

For loops over negation we depend on the current XSB support. XSB supports loops over negation in dynamically stratified programs (Swift et al. 2013).

It is necessary to add the flag `:-set_prolog_flag(unknown,fail).` to our Prolog code because it is possible to have no true tuples in the interpretation of open input predicates in a $\mathrm{FO}(\cdot)^{\mathrm{IDP3}}$ model. This would result in an XSB program that has no fact for that predicate. Standard XSB considers this a programming error. Setting this flag means that XSB returns `fail` for queries to predicate symbols that have no rules or facts.

We implemented the evaluation of the input∗ predicates as a separate step in IDP3, because it is also useful for evaluating some class of output predicates.

4.2 Further Uses of XSB

Our current implementation is limited to input∗ predicates, but we can use our method in more cases. The transformation steps remain the same.

For monotone definitions, we can relax the requirement that the input predicates need to be given completely in the structure. We simply use their *partial* interpretations in our method and the computed answers can be used as their partial interpretation. We keep the definitions in the theory. This extension can easily be added to the implementation.

Another observation is that there are also definitions that contain open predicates whose interpretation is decided by the search component of IDP3, *non-input* open predicates. These non-input open predicates currently block the use of our method. However, we can construct residual programs with XSB. Consider the example given in Figure 2.

In a sense what we do is a kind of partial evaluation or program specialization. To

```
:-table p/2, b/1.     % a/1 is an input predicate, b/1 is non-input
p(X,Y) :- a(X), b(Y).
a(1).
a(2).
b(Y) :- tnot b(Y).
```

The residual program for `?- p(X,Y).` is:

```
p(1,Y) :-  b(Y).
p(2,Y) :-  b(Y).
```

Fig. 2. Example of a residual program

implement this, we need to replace the original definitions by the residual ones which are special cases of the transformed ones. We are currently investigating this.

Another application of our method is in the context of LUP. For each FO($\cdot$) sentence, LUP can be represented symbolically by a set of recursive rules. All of these rules together form an FO($\cdot$) definition (Vlaeminck 2012). This definition models the dependencies between the truth values of subformulas and allows derivation of useful information (about the so-called bounds) by taking into account information available in the structure. This is a different approach w.r.t. (Wittocx et al. 2010) (which is an approximative method), but similar to (Vaezipoor et al. 2011). Preliminary experiments show that our method can be used for computing the definitions produced by this form of LUP, but it seems important to use a good clause ordering.

Our method can be considered as a special case of this LUP: the structure-specific information is given by the open input predicates and the corresponding input$*$ predicates are computed. The above paragraph about monotone definitions can also be considered a special case of LUP.

5 Experiments

In this section, we compare our method IDP3 $^{\texttt{XSB}}$ with the current version of IDP3 and with the state-of-the-art systems CLINGO and DLV. Figure 3 contains the results of our experiments performed on an `Intel(R) Core(TM) i5-3550 CPU @ 3.30GHz` with a cutoff of 500s. Experiments that exceeded the cutoff are marked with "-". We run experiments with gringo version `4.0-rc2`, clasp version `2.1.1` and the DLV build of `Dec 17 2012`. The tools needed to run these experiments can be found on `http://dtai.cs.kuleuven.be/krr/research/experiments`.

Experiments were performed for REACH (reachability with undirected edges), PATH10 (calculate all pairs of nodes that are connected by a path consisting of 10 nodes), HP (Hamiltonian Path), NQ (NQueens) and HNQ (NQueens that computes also which placed queens are able to hit each other if they could also move like knights). All running times are given in seconds. We show for each system the grounding time (first column) and the total solving time (second column), except for DLV for which we only show the solving time. The second column in Figure 3 indicates what portion of the predicates are input$*$ predicates. For example, i/j indicates that there are a total of j predicates and i of those are input$*$. The third column lists the problem sizes and they mean the following: for graph problems (REACH, PATH10, HP), the problem size is the

Fig. 3. Comparison of the grounding and total execution times.

Problem	# input∗	Size	IDP3		IDP3 ^{XSB}		CLINGO		DLV
			ground	solve	ground	solve	ground	solve	solve
REACH	2/2	10	0.06	0.06	0.09	0.09	0.01	0.01	0.01
		40	0.24	0.24	0.13	0.13	0.01	0.01	0.02
		400	106.04	106.08	1.50	1.50	0.27	0.44	4.09
PATH10	1/1	50	0.18	0.24	0.09	0.09	0.01	0.01	0.01
		300	6.00	6.00	0.25	0.25	0.01	0.01	0.04
HP	0/3	40	0.09	0.09	0.09	0.09	0.01	0.01	0.05
		400	4.15	5.00	4.15	5.00	0.03	0.15	9.34
NQ	2/3	5	0.18	0.18	0.09	0.09	0.01	0.01	0.01
		20	0.78	0.78	0.13	0.13	0.01	0.01	0.09
		100	226.07	229.09	0.39	0.58	0.10	0.20	-
HNQ	4/5	5	0.32	0.33	0.15	0.16	0.01	0.02	0.02
		20	1.52	1.53	0.80	0.18	0.02	0.03	0.23
		50	39.80	40.10	24.80	0.79	0.08	0.17	-
		100	-	-	-	4.08	0.36	0.64	-

number of nodes in the graph (increasing graphs have a constant edge density) and for board problems (NQ and HNQ), the problem size represents the board size.

For every problem except for HP, our method results in a significant decrease in running time compared to IDP3. Experiments also show that the running time complexity is of a lower order: larger problems benefit from larger speedups of our method. For HP running times stay the same because in this problem no input∗ predicates are used.

Note that the solving time ($N = 50$: 0.79) is better than the grounding time ($N = 50$: 24.80) for the HNQ problem with the IDP3 ^{XSB} system. This is caused by the output predicates that are present. The grounding of output predicates (in this case, the queens that can hit each other by moving as knights) can, in the context of model expansion, be delayed until an interpretation for the remainder of the predicates (the placement of the queens) has been found. When we perform the grounding operation as a standalone operation with IDP3 ^{XSB}, these output predicates cannot be delayed and are thus also grounded, since no interpretation for the remainder of the predicates is computed. For the information in the second column, the output predicates are considered to be input∗ predicates, since we also use `XSB` to compute their interpretation.

Experiments show that IDP3 ^{XSB} results in large speedups with respect to IDP3 for problems where input∗ predicates are used. Our results are also comparable to other ASP systems that use the semi-naive bottom-up approach, such as CLINGO and DLV. To get an idea of the communication overhead between IDP3 and `XSB`, we ran programs with the transformed rules directly with `XSB` and compared it with calling them from within IDP3 ^{XSB}. These experiments showed that the communication between the systems causes an increase of about 30% in time spent computing the definitions.

6 Conclusions

The grounding phase is well studied in the context of Answer Set Programming (Faber et al. 2012; Gebser et al. 2007; Syrjänen 1998; Wittocx 2010). The grounder transforms the input program into a semantically equivalent one with no variables and tries to avoid the combinatorial explosion that arises by naively instantiating the atoms in the program

by all their instances. DLV and gringo (CLINGO's grounder) ground using an instantiation algorithm that is based on the well-known semi-naive bottom-up computation. They assure groundings only contain ground atoms that can be derived from the program. Moreover, ground rules are simplified by removing literals known to be true. As a consequence, the intentional predicates in safe (normal, i.e., deterministic) stratified programs are completely evaluated.

These completely evaluated predicates can be seen as our input$*$ IDP3 predicates. IDP3 is a model generator for FO$(\cdot)^{\texttt{IDP3}}$ which extends first-order logic with inductive definitions and allows the programmer to write declarative specifications for his problems. For the programmer it becomes easier to give the specifications, but FO$(\cdot)^{\texttt{IDP3}}$ requires additional intelligence during the grounding. The grounder of IDP3 uses a form of LUP to derive extra information that can be used to reduce the grounding. LUP is effective (Wittocx 2010), but input$*$ predicates need special treatment. In this paper we use `XSB` and its tabling to compute the interpretations of input$*$ predicates whose first-order bodies are transformed into Prolog. Note that the safeness requirement is not needed as variables in IDP3 are typed and as such their values are known.

The paper (Faber et al. 2012) discusses a number of optimization techniques in context of the DLV system. Some of them are relevant for our conjunctive bodies. Their program rewriting (Faber et al. 1999) strategy pushes projections and selections down the execution tree, while their body reordering criterion (Leone et al. 2001) takes into account the impact on the reduction of the search space and tries to detect inconsistencies early by preferring literals with bounded variables. We currently use a simple body reordering method. Other optimizations such as Dynamic Magic Sets (Alviano and Faber 2011) are related to the bottom-up strategy, while we use the tabled top-down evaluation of `XSB`.

Experiments show the proposed method results in large speedups with respect to IDP3 for problems where input$*$ predicates are used. Our running times are comparable to other ASP systems that use a semi-naive bottom-up approach, such as CLINGO and DLV, despite the communication overhead between IDP3 and `XSB`.

One feature is not yet supported by our transformation: recursive aggregates. An aggregate is recursive if, as part of a body of a rule in a definition, its set expression contains a recursive call.

By coupling IDP3 and `XSB` we can further explore the tabled evaluation to deal with partial open predicates and the use of residual programs as a form of partial evaluation. Moreover, the propagating definitions generated for LUP (Vlaeminck 2012; Vaezipoor et al. 2011) can also be transformed into `XSB` programs. In this way the grounder of IDP3 can further be improved.

Another interesting issue is whether, as literals in the theory get more instantiated during the search phase, it would be interesting to evaluate these propagating definitions for these literals. This evaluation then computes the newly propagated information following from the choices the solver made. This form of goal-directed evaluation will probably require a better integration between IDP3 and `XSB` to reduce communication overhead and to benefit from the support for incremental tabling of `XSB`.

References

Aavani, A., Wu, X. N., Tasharrofi, S., Ternovska, E., and Mitchell, D. G. 2012. Enfragmo: A system for modelling and solving search problems with logic. In *LPAR*. 15–22.

Alviano, M. and Faber, W. 2011. Dynamic magic sets and super-coherent answer set programs. *AI Commun. 24*, 2, 125–145.

Blockeel, H., Bogaerts, B., Bruynooghe, M., De Cat, B., De Pooter, S., Denecker, M., Labarre, A., Ramon, J., and Verwer, S. 2012. Modeling machine learning and data mining problems with FO($\cdot$). In *Proceedings of the 28th International Conference on Logic Programming - Technical Communications (ICLP'12)*, A. Dovier and V. Santos Costa, Eds. Schloss Daghstuhl - Leibniz-Zentrum fuer Informatik, 14–25.

De Pooter, S., Wittocx, J., and Denecker, M. 2011. A prototype of a knowledge-based programming environment. In *International Conference on Applications of Declarative Programming and Knowledge Management*.

Eén, N. and Sörensson, N. 2003. An extensible SAT-solver. In *SAT*, E. Giunchiglia and A. Tacchella, Eds. LNCS, vol. 2919. Springer, 502–518.

Faber, W., Leone, N., Mateis, C., and Pfeifer, G. 1999. Using database optimization techniques for nonmonotonic reasoning. In *INAP Organizing Committee DDLP'99*. 135–139.

Faber, W., Leone, N., and Perri, S. 2012. The intelligent grounder of DLV. *Correct Reasoning*, 247–264.

Gebser, M., Kaminski, R., König, A., and Schaub, T. 2011. Advances in *gringo* series 3. In *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, J. Delgrande and W. Faber, Eds. Lecture Notes in Artificial Intelligence, vol. 6645. Springer-Verlag, 345–351.

Gebser, M., Schaub, T., and Thiele, S. 2007. GrinGo : A new grounder for answer set programming. In *LPNMR*, C. Baral, G. Brewka, and J. S. Schlipf, Eds. LNCS, vol. 4483. Springer, 266–271.

Leone, N., Perri, S., and Scarcello, F. 2001. Improving ASP instantiators by join-ordering methods. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*. LPNMR '01. Springer-Verlag, London, UK, UK, 280–294.

Mariën, M., Wittocx, J., Denecker, M., and Bruynooghe, M. 2008. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In *SAT*, H. Kleine Büning and X. Zhao, Eds. LNCS, vol. 4996. Springer, 211–224.

Mitchell, D. G. and Ternovska, E. 2005. A framework for representing and solving NP search problems. In *AAAI*, M. M. Veloso and S. Kambhampati, Eds. AAAI Press / The MIT Press, 430–435.

Swift, T. and Warren, D. S. 2012. XSB: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming 12*, 157–187.

Swift, T., Warren, D. S., Sagonas, K., Freire, J., Rao, P., Cui, B., Johnson, E., de Castro, L., Marques, R. F., Saha, D., Dawson, S., and Kifer, M. 2013. *The XSB System Version 3.3.x Volume 1: Programmer's Manual*.

Syrjänen, T. 1998. Implementation of local grounding for logic programs with stable model semantics. Tech. Rep. B18, Helsinki University of Technology, Finland.

Vaezipoor, P., Mitchell, D. G., and Mariën, M. 2011. Lifted unit propagation for effective grounding. *CoRR abs/1109.1317*.

Vlaeminck, H. 2012. Applications of feasible inference for expressive logics. Ph.D. thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium.

Wittocx, J. 2010. Finite domain and symbolic inference methods for extensions of first-order logic. Ph.D. thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium.

Wittocx, J., Denecker, M., and Bruynooghe, M. 2013. Constraint propagation for first-order logic and inductive definitions. *ACM Transactions on Computational Logic*. Accepted.

Wittocx, J., Mariën, M., and Denecker, M. 2010. Grounding FO and FO(ID) with bounds. *Journal of Artificial Intelligence Research 38*, 223–269.